User Manual: Type System

Abstract:

The type system of AutoFOCUS describes the data modelling language with the possibilities of defining and inferring types in AutoFOCUS models. This description includes the concepts, grammars and inference rules and relevant subsets. Furthermore open issues (implementation bugs) are described.

The type system of AutoFOCUS has been developed by Validas AG within the BMBF project AutoMODE in cooperation with TU Munich and ETAS GmbH.

Contents:

1 Introduction 4

2 Basic Concepts and Definitions 5

2.1 Terms 5
2.2 Types 6
2.3 Type Classes 7
2.4 Module Concept 8
2.5 Implementation Types 9

3 Using Type System 13

4 Type Inference 16

4.1 Context of Type Inference 16
4.2 Atomic Rules 17
4.3 Propagation 18

5 Syntax 19

5.1 Terms 19
5.1.1 Constants 19
5.1.2 Variables 19
5.1.3 Applications 20
5.1.4 Sets 21
5.1.5 Arrays 22
5.1.6 User Definitions 23
5.2 Types 25
5.2.1 Type Terms 25
5.2.2 Data Definitions 25
5.2.3 Type Abbreviations 28
5.2.4 Implementation Types 28
5.2.5 Range Types 29
5.2.6 Casts 29
5.3 Modularity (Import) 29

6 Models 30

6.1 Terms 30
6.2 Types 33
6.3 Type Classes 35
6.4 Modules 37

7 Predefined Elements 39

7.1 Constants 39
7.2 Functions 39
7.3 Types 40
7.4 Casts 41
7.5 Classes 42

8 Open Issues 43

9 References 44

A. AutoFOCUS Grammar 45

Introduction

This documentation describes the type system of AutoFOCUS [AF]. It has been developed from Validas AG within the BMBF-Project AutoMODE and is currently integrated into the tool AutoFOCUS2 [AF2] from TU Munich. While AutoFOCUS supports different type systems for different purposes by allowing to create different projects with different languages. Currently the supported languages are Java, Quest, AutoMODE. In AutoFOCUS2 there is only one language realized. In this document we describe the type system of AutoFOCUS which is the type system of AutoFOCUS2 and identical to the type system of the AutoMODE language projects in AutoFOCUS.

The main design goals for the languages was to have a well defined, simple but also expressive and functional language as sound basis for modelling embedded systems.

This document describes the expression languages of AutoFOCUS including the main feature of type inference and checking. It is intended to serve as an introduction for advanced AutoFOCUS modellers (AutoFOCUS models can also be build without user defined types and functions of this language), and also for developers which want to write their own model based analyses or generators for specific targets.

Note that many features like indexed arrays, range types, type class instantiations, sets have not been tested intensively, such that there might occur bugs.

This manual is structured as follows: Section 1 introduces the basic concepts informally, Section 2 shows how to use the type system. Section 3 informally describes the inference rules. In Section 4 the syntax of AutoFOCUS is explained. Advanced concepts, not explained in the introduction, are also described Section 4.Section 5 describes the meta models of the type system. Section 6 lists the predefined elements. The manual closes with open issues and references (Section 7 and Section 8).

1 Basic Concepts and Definitions

AutoFOCUS is strongly typed, i.e. every term in the model has a fixed type. The type must not be totally fixed ("**monomorph**"), but can be **polymorphic**. For example the polymorphic type `NUM` stands for arbitrary numeric values and can be instantiated by monomorphic types like `Int`, `Float` or `uint8`.

1.1 Terms

Values are the informations that are exchanged by AutoFOCUS. Values can be default values (like `1`, `True`, `{1,2}`) or they can be constructed using **constructors** from other values (e.g. the `{` and `}` for constructing sets) . Constructors for a user defined type can be defined in user defined type definitions. Constructors can also have no arguments, in this case they construct the values without arguments (like constant functions without arguments). Constructors can be used in pattern matching (see Section 4.1.6).

Terms are the descriptions of values. Examples of terms are `1`, `True`, `2+3`, `three` and `f(x)`. Every value is a term, but also user defined constants, variables and functions can be used in terms. A **variable** can be instantiated by concrete values (if type of the variable and the type of the value fit together). AutoFOCUS uses variables for modelling input of the components (e.g. the `x` in the input pattern `ein?x` is a variable) and within function definitions. A **function** is applied to it's arguments (values) and computes a new value. The number of arguments of a function is called it's **arity**. Examples for functions are `+` (with arity 2) and `f` (with arity 1 in the case of `f(x)`). The user can define constants (like `three = 1+2`) and functions (like `f(x) = x+1`) (see Section 4.1.6). Section 5.1 describes the models that are used to represent terms in AutoFOCUS.

A **user defined constant** and a **user defined function** has a definition from the user. All other constants and functions are called **default**

constants and **default functions**. Examples for default constants and functions are **`True`**, **`1`** and **`+`**.

1.2 Types

Every term in AutoFOCUS has a type. A **type** describes a set of possible values.

The user can define a type for a term and the types can be inferred. For example the type of **`True`** is **`Bool`** (which describes the possible values **`True`** and **`False`**) and the type of **`1`** is **`NUM`** (which describes all numeric values). The user can define the type of a term by writing type annotations (for example **`1:Int`**), or by defining signatures for functions (e.g. **`max:NUM->NUM`**). In this case there is not only a type, but also a defined type. Note that defined types can also be more general than the inferred types, especially in component models where types are specialized by propagating between connected components.

Types can be constructed (like terms) from type constants, type variables and type building functions (**type constructors**). Examples for type constants are **`Bool`** and **`Int`**. An example type constructor is set, which is denoted by curly brackets (**`{}`**). It can be used to construct the types like **`{Int}`** or **`{Bool}`**. The number of arguments a of a type constructor is also called it's **arity**. A **type variable** can be instantiated by type terms (if the type classes of the type fit together). Type variables are used in the definition of types, e.g. by defining the type of lists of arbitrary elements.

To avoid misspelling errors in types for example by typing **`imt8`** instead of **`int8`**, type variables must start with a "´", i.e. if a port shall have the general type which can be matched with any other type, the type name for this port must be **`´a`**.

Note that AutoFOCUS has the functional type constructor **`->`** for describing the types of functions. For simplicity values must not have functional types. The arity of **`->`** is two, however it can have also more arguments, for example the type of **`+`** is **`NUM,NUM -> NUM`**.

1.3 Type Classes

Type classes are used for defining functions that operate on different types. For example the function that checks if an elements is in a list of arbitrary elements should not have the type **`list(a) -> Bool`** but the type **`list(EQ)-> Bool`**. **`EQ`** requires that the instantiated types provides an equality operation for comparing elements.

A **type class** is a set of types. For simplicity AutoFOCUS supports only the following three type classes.

- **`EQ`**
- **`ORD`** is a specialization of **`EQ`**, i.e. all types in **`ORD`** belong also to **`EQ`**
- **`NUM`** is a specialization of **`ORD`**.

If a type has no class it does not belong to any of these classes. This is the most general case.

Every type class is characterized by the presence of some (characteristic) operations that can be used for defining operations on that type. The characteristic operations for the type classes in AutoFOCUS are:

- **`EQ`**:
  - equal: **`== : EQ,EQ -> Bool`**
  - inequal: **`!=: EQ,EQ -> Bool`**
- **`ORD`**
  - less: **`< : ORD,ORD -> Bool`**
  - less or equal: **`<=: EQ,EQ -> Bool`**
  - greater: **`> : EQ,EQ -> Bool`**
  - greater or equal: **`>=: EQ,EQ -> Bool`**
- **`NUM`**
  - addition: **`+ : NUM,NUM->NUM`**
  - subtraction: **`- : NUM,NUM->NUM`**
  - multiplication: **`* : NUM,NUM->NUM`**
  - division: **`/ : NUM,NUM->NUM`**

If a type belongs to a type class, it provides the characteristic operations of that class. For example the type **`Bool`** only belongs to the type class **`EQ`**,

e.g. terms like **`True == X`** are correctly types, while **`True>x`** or **`True+False`** are not correct.

User defined types can be **instantiated** into a specific type class in two ways:

1. by automatically **deriving** the characteristic functions from the order of the constructors and the characteristic operations of the used types.
2. by manually defining the characteristic functions. This features is currently not supported in AutoFOCUS, but the models and the syntax are present such that it can be integrated, if this features is needed.

1.4 Module Concept

The user defined definitions of types and functions can be modular. The "Symbol Table" (represented by the DTD in [AFMM]) of AutoFOCUS-components consists of all defined functions and constants in the DTDModules. Every DTDModule corresponds to one textual document in AutoFOCUS (called DTD).

The module concept of AutoFOCUS allows to use qualified names of the form **`<DTDModuleName>.<element>`**, and to import elements from other DTDModules, such they can be used without qualification in the DTDModule.

Since AutoFOCUS components cannot declare import statements for referencing elements from DTDModules they have to use qualified names, if the referenced elements are not unique in the symbol table. For example if two constants **`max`** exist in two DTDModules **`M1`** and **`M2`** the components have to refer them as **`M1.max`** and **`M2.max`**, while the DTDModule **`M3`** can refer **`max`** without qualification as **`max`** if it has been imported by declaring **`import M1.max; or import M2.*;`** within the DTDModule.

1.5 Implementation Types

The concept of **implementation types** allows to refine numeric types. The goal of using implementation types is to validate the models numeric properties as it will be computed on target hardware with restricted numeric features. For example many processors do not support floating point arithmetic but only integer arithmetic. Therefore the floating point values of the model have to be replaced by fixed-point approximations on the target. Implementation types allow to model and simulate these approximations within AutoFOCUS and support therefore the detection of numeric errors within the model. For this purpose there are two different simulations in AutoFOCUS integrated: the ordinary simulation, which used the defined types and the **implementation simulation**, which uses the implemented types (see Figure 1 for starting the implementation simulation).

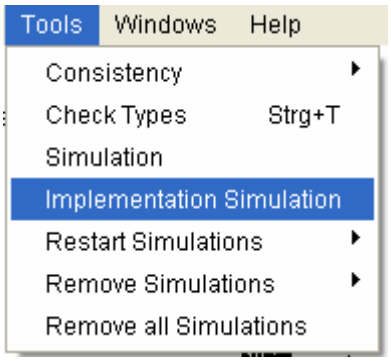

Figure 1: Starting Implementation Simulation in AutoFOCUS

Mapping AutoFOCUS integers (with unrestricted size) to integers of fixed size is a similar challenge.

Another benefit of implementation types is that general models can be restricted to small ranges. This allows to verify the model using model checking (Note that currently the model checking extension does not explore this feature).

Using implementation types consist of two parts:

1. Definition of fixed point types. This can be omitted, if the predefined integer types (see Section 6.3) are used.
2. Mapping the general types to fixed point types

It suffices to specify the mapping for the ports and local variables in the system. The other used elements (terms, constants and functions) are inferred from the type system.

The mapping has to be defined using so called **implementations**. An implementation consists of type implementations (called TypeImpl in [AFMM]) that map (abstract) types to (concrete) types. Implementations are defined in the data type definition. An example for an implementation is the following definition:

```
implementation myImpl = [Float -> int32, TT -> uint8];
```

Note that implementations can be extended by existing implementations using the operators +, such that `Fine = [Int -> int32] + myImpl;` is also a valid implementation.

For every port and local variable in the implemented model one mapping has to be selected. If there is more than one mapping (for the type of the port/variable) defined a selection box appears that allows to choose the desired implementation (see Figure 2). The resulting implemented type is displayed using "-> type" in the selection box.

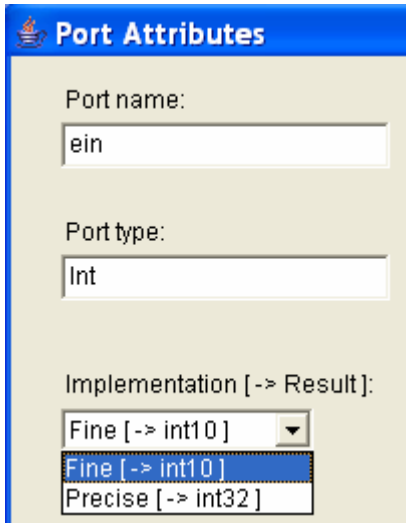

Figure 2: Selecting an Implementation for an AutoFOCUS-Port

If implementation types are used the type checker infers the implemented types. In the model the implemented type is displayed as [-> type], see Figure 3.

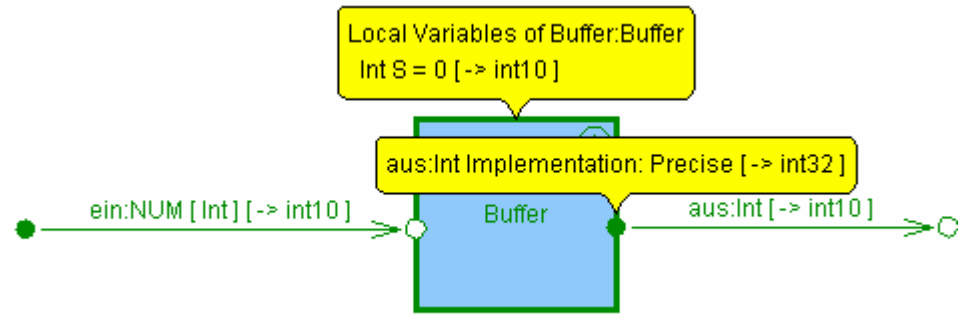

Figure 3: Display of Implemented Types in SSDs

SSDs have to be reusable in different context (Sharing) with different implementations. This requires that once a type of Port **`ein`** is implemented using **`Fine`** and in another context using **`Precise`** SSDs have implementation parameters. An **implementation parameter** is declared in the SSD like a parameter or a local variable (see Figure 4). Like parameters the implementation parameters are available in all subcomponents.

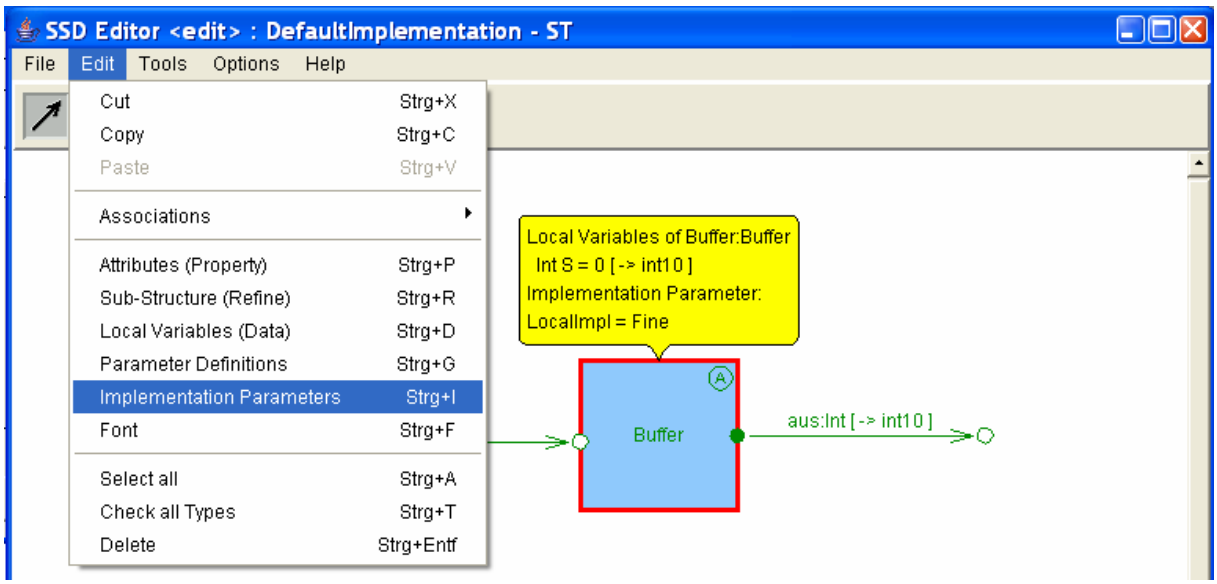

Figure 4: Implementation Parameters

Since it might be much work to specify an implementation for every type separately, AutoFOCUS supports two helpful concepts:

1. If only one implementation is applicable to the type this mapping is used.
2. If there exists an implementation called "**`Default`**" this implementation is selected.

The computation of the implemented type for a port or local variable is done as follows:

1. select the implementation mapping as follows:
    a. the user has explicitly selected an implementation, or
    b. there exists a Default-Implementation.
2. The available implementations are searched bottom-up from the current component until the top component
3. compute the implemented type from the mapping by finding the TypeImpl (pair of types) for which the abstract type matches the type of the port. The concrete type of the TypeImpl is the implemented type of the port/variable.

Within the API of AutoFOCUS there are three different functions to access the different types.

- getDefinedTypeModel: returns the user defined type
- getType: returns the inferred type from the type checker
- getImplementedType: returns the type from the implementation. If there is no implementation available the original type is returned.

Note that these functions can only return defined values, if the type checking has been applied.

2 Using Type System

The type system of AutoFOCUS is an advanced consistency check of the models. The terms used within the models have to be type correct. The operation to ensure that the model elements are type correct is called **type check**. Not that the semantics of AutoFOCUS is only defined for type correct models, i.e. incorrect models cannot be treated. Therefore the type check is executed before all operations that base on the semantics of AutoFOCUS (like simulation, code generation and model checking).

In addition to this build in checks the type checker can be started manually for the complete project, or for some parts of the model (components, automata, transitions, DTDs).

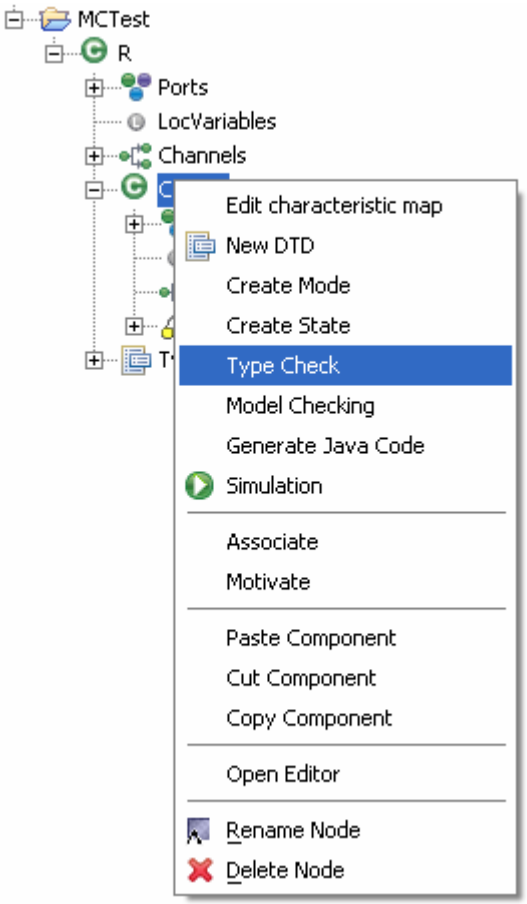

Figure 5: Starting Type Check Manually

Figure 5 shows the start of the type check in AutoFOCUS2. The displayed menu appears if a component is selected and the right moue button is pressed.

In AutoFOCUS the type checker can be started within SSD using the key combination Strg+T. Within STDs the types are checked permanently and the results are displayed using (configurable) colors. Figure 6 shows the results of an incorrect transition model in AutoFOCUS.

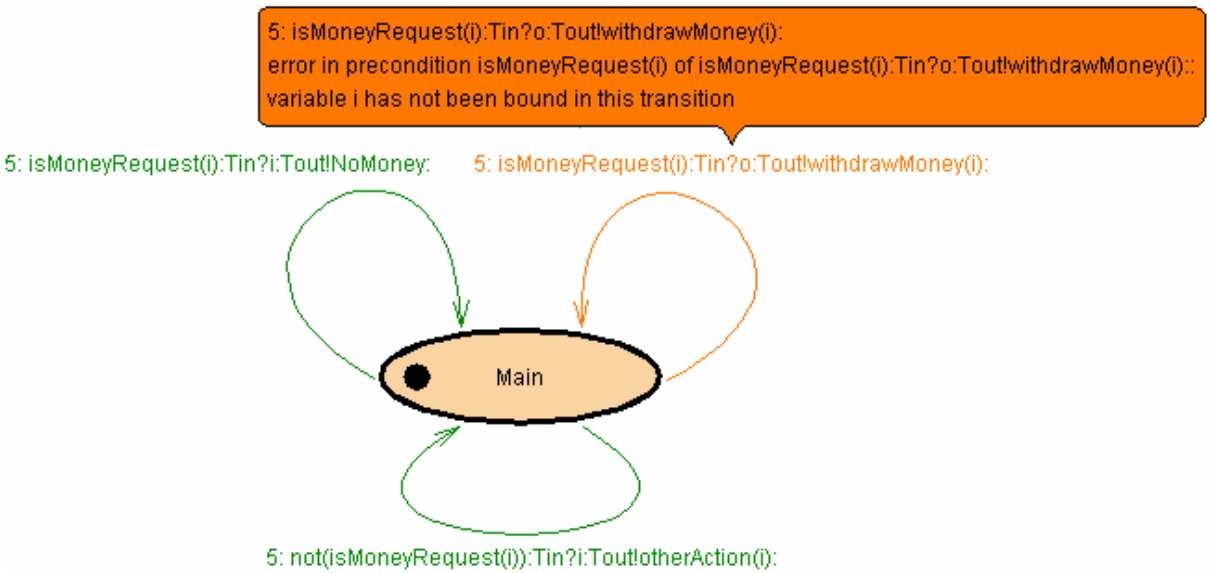

Figure 6: Results of Type Checking in AutoFOCUS

The types of the components in AutoFOCUS can be defined at the ports (and the channels) of the components. If the defined type is more general than the inferred type, the inferred type is displayed in square brackets. For example of if we define a channel with a generic type in AutoFOCUS using the type variable `` `a `` (type variables are prefixed using a `` ` ``) which is connected to a Port with a concrete type **`Int`**, than the type **`Int`** is inferred for the channel and the connected ports and displayed in square brackets (see Figure 7).

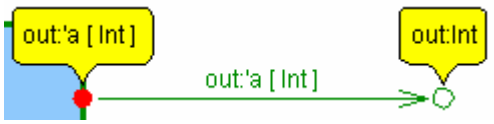

Figure 7: Inferred Types in SSDs

Note that the inferred types within the components (e.g. from the Automaton) are not propagated into the SSDs in AutoFOCUS. Hence the

defined types of the ports of the component are used for checking the complete system. This modularity feature of the type checker in AutoFOCUS allows to type check large models without checking all subcomponents. Therefore the types of the components should not be more general than the used types within the components. In AutoFOCUS2 the type checking is deep, i.e. includes all subcomponents and their descriptions.
If the types are not propagated as expected a complete type check of the project is recommended.

3 Type Inference

Type inference assigns concrete types to the untyped variables and applications in a term, e.g. type inference of **`x==1`** assigns the type **`NUM`** (from the constant **`1`**) to the variable **`x`** and the type **`Bool`** to the term consisting of the application of **`==`** to **`x`** and **`1`**. Similar inferences are applied to infer the types and type classes of type variables.

The type context stores the inferred types of the variables and type variables. The results of the type inference are stored in the model (mostly in the Type-Field), such that they can be displayed and used in all model based applications of type checked models. The results are only stored in variables and applications (since the constants are global and can be used in different contexts their inferred types must not be stored at the constants. The inferred types of constants are stored in the type contexts and are set in the applications using the constants.

3.1 Context of Type Inference

The type inference depends on the context. A component with a specific type can be type correct in one context, but may not fit into another context. If the type check is started in a component, only this component is checked, but not it's context. If a project is type checked, all top components in the projects are checked.

Another important feature for the type inference is the binding mechanism of type variables. Type variables are identified only within one component description), hence a (shared) component with a type using the variable 'a can have different types in different instantiations in different components. Within one component type variables with the same name (e.g. 'a) are identified. Note that type variables NUM, EQ and ORD are not identified, such that components with type classes can be used in different instantiations within the same component (see example in Figure 8).

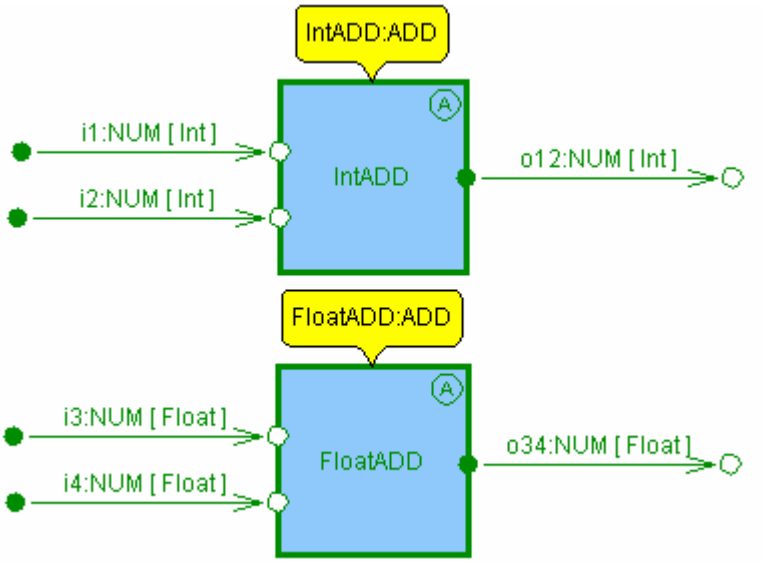

Figure 8: Mixing Types in AutoFOCUS

The context of the type checking in DTDs (Data Type Definitions) is only relevant for checking user defined functions. Every function patterns has it's own context and the resulting type for the function is combined over all patterns. For example a function

```
fun test(True,x) = x
  | test(x,0) = 1;
```

In the first line the inferred type for the variable **x** is a type variable **'a** and the type of **test** is restricted to **Bool,'a -> 'a**. In the second pattern the type of the different variable **x** is inferred to another type variable **'b**. the type of the function **test** is restricted to **'b,NUM->NUM**. combining it with the first type of **test** the resulting type for **test** is **Bool,NUM->NUM**. This is the defined type of the function test. If the type of test should be more specific a more concrete signature definition can be added, e.g.

```
test:Bool,Int->Int;
```

In this case the defined type of **test** is set to the specified type **Bool, Int->Int**.

3.2 Atomic Rules

The types of constants (and functions) are inferred from their definitions. A constant can be defined as a constant (or function), than the defined type is inferred as described in the previous section. If a constant (or

function) is a constructor of a data type, the type is directly used from the type definition.

Default constants are constants (that have no user defined definition) are assigned to types as follows:

- **`True:Bool`**, **`False:Bool`**
- Numeric values (without comma) are numeric values (**`NUM`**)
- Numeric values (with comma) are **`Float`** values.

3.3 Propagation

The propagation of types in terms ensures that the most general type of the terms is inferred. The most general types are stored in application terms (which are separate objects) and variables (which are unique within one term).

The propagation of types identifies the types of variables in all occurrences of the term and the arguments of functions with the expected types. Furthermore type annotations are integrated into the propagation. For example type checking the term **`x+(y:Int)>0`** infers the type **`x:Int`** (from the type annotation), **`y:Int`** (from the application of **`+:NUM,NUM->NUM`** to the **`y:Int`** and **`0:Int`** from the result of **`>:ORD,ORD->Bool`** applied to the **`Int`** result of the addition.

4 Syntax

The actual syntax of the AutoFOCUS can be seen in the help grammar menu of the DTD Editor. In the following the most important rules and the related concepts (if not explained in Section 1) are explained.

4.1 Terms

There can be three different terms in AutoFOCUS models: constants, variables, and applications (like in the lambda-calculus). These are described in the following subsections. Sets and arrays are special form of terms and are described separately.

4.1.1 Constants

Constants are referred by their names (or directly by the values). The following rules of the AutoFOCUS grammar describe the definition of constants in AutoFOCUS terms:

```
term12 = sign neg* qid
       |  char
       |  string
       |  sign? neg? number
```

Where **qid** are possibly qualified identifiers (see Section 1.4). **char** refers to characters (‘**c**’), **string** refers to arbitrary text (without escape characters) included in quotes (“**hello world**”) and **sign** can be **+** or **–**. Negation can be formulated using **!** (or **neg**). Numbers can be defined as decimal or hexadecimal numbers according to the following rules:

```
number = decimal | hexnumber;
decimal = digit* '.'? digit+;
hexnumber = ('0x'|'0X') (digit | ['A'..'F'] | ['a'..'f'])+;
```

4.1.2 Variables

Variable terms are build by their names. E.g. **x**. The scope of the variables depends on their context. This can either be a function definition which uses variables, or can be a transition.

The grammar for variables is the same as for constants, e.g. variables are constants with names that have be bound in a context.

```
term12 = sign neg* qid
```

4.1.3 Applications

Application terms consist of a function (called "header") and their arguments. The header function is modeled as a constant. The arity of the header has to be equal to the number of arguments of the application. Applications can be build with the build-in infix operations (e.g. **`1+x`**) or with prefix-notation (e.g. **`f(x,y)`**). Infix operations use the usual binding priorities to reduce the number of required brackets in terms like **`2<1+2*3`**. The build-in functions and their priorities are described in Section 6.2.
The infix rules form a large part of the AutoFOCUS grammar, all follow the same scheme. The following rules show some examples.

```
term10 ::= term11 '+' term10
         |  term11 '-' term10
         |  term11.

term11 ::= term12 '*' term11
         |  term12 '/' term11
         |  term12 '%' term11
         |  term12.
```

The grammar for the prefix notation requires an identifier followed by brackets with the function arguments separated by comma:

```
term12 =  id  '(' term coterm* ')'
coterm = ',' term;
```

In addition to the infix and prefix terms some special mixfix application terms can be build: if-then-else, let-terms, sets and arrays. Furthermore there are some special terms (like case, or merge in the grammar, however they are used for special purposes and therefore cannot be used in all situations). The set and array terms are described in the following sections. The rules for special terms are:

```
term12 =  'if' tterm 'then' tterm 'else' tterm 'fi'
```

```
            | 'let' id '=' arrterm 'in' term12
```

The let term is a special term which allows to assign variables in a sequential way. For example in the function definition

```
fun f(x) =      let y = x+x in
                let z = y*y in (z+y+1);
```

4.1.4 Sets

AutoFOCUS allows to build (finite) sets of elements. All elements in the set have to be of the same type. In contrast to weakly typed languages, the elements in the set must be of the same monomorphic type. Sets like `{1,True}` are forbidden.

Sets can be constructed using set brackets (`{` and `}`). The curly brackets are also used to denote the type of sets. E.g. the type `{Int}` is the type of set of integers. Furthermore it is possible to add, subtract sets using `+` and `-`. `*` is used for intersection of sets. The test whether an element is in a set is expressed using the infix operation `in`, e.g. `3 in {1,2,3}`. Furthermore subsets with predicates can be formulated using `|`, e.g. `{x | x>1,x<10}`. On the left side of the `|` pattern matching can be applied to define variables for the right side. E.G. `{x:Bool | x=True}`, or `{ IPair(x,y) | x !=y }`. Note that with this notation also infinite sets could be specified (e.g. the pair example, if `IPair` is defined as a general data constructor `data IntPair = IPair(Int,Int)`). A set is **finite**, if it consists only of finite (enumerated, boolean or constructed from user defined constructors over finite types) data types. Note that the operators `any` and `rnd` can be inefficient, if restrictions are used, since they base on a simple generate and check approach. For example the value `any({ BPair(x,y) | y!=x })` is (using the Boolean Pairs `data BoolPair = IBPair(Int,Int)`).) computed as follows: `BPair(False,False)` is generated and checked for `False!=False`. The next generated value is `BPair(True,False)`, which satisfies `True!=False` and is therefore the result of the `any` operation. `rnd` uses random values instead of

incremental generation. Functions over finite sets (e.g. the sum of the values in the set) can be build using the **any** operator and the difference.

```
fun sumOfValues(s) =
if s=={}
then 0
else  let x = any(s);
      in x + sumOfValues(s-{x})
fi;
```

To make the specifications of set functions more readable, they can be formulated using pattern matching with a special pattern. For example:

```
fun sumOfValues({}) = 0
        |  sumOfValues({x} + s) = x+ sumOfValues(s);
```

The rules in the grammar for defining sets are (apart from the application of functions, which is described in Section 4.1.3)

```
term12 ::= ..
        |  '{' '}'
        |  '{' tterm coterm* '}'
        |  '{' tterm '|' tterm coterm* '}'
tterm ::= interm typean?.
interm ::= arrterm 'in' arrterm
        |  arrterm.
```

The rules for **tterm** and **arrterm** denote that the infix operator **in** has the lowest priority (since all other infix operators are defined in the rules for the non-terminal arrterm).

4.1.5 Arrays

Arrays are structures (of fixed size) of similar values that can be accessed using indices. In AutoFOCUS the default array index is the type Int (starting with the index `0`), however other index types can be used. Arrays can be declared. In this case the type is constructued by square brackets (`[,]`). E.g. an arrays of Boolean values is declared as follows: **`ba:[Bool]`**. If the index type shall also be declared, this can be done using a comma. E.g. the above declaration is equivalent to **`ba:[Bool,Int]`**. The access to

the values is per index, e.g. **ba[0]** or **ba[1]**. Arrays can be constructed using the square brackets, e.g. **[4,5,6]**. Furthermore arrays can be defined with index values, e.g. **[1=5,0=4,2=6]**. This case can be useful if other index types are desired. Note that the indices must be a sequence, i.e. no value must be missing like in **[1=5,4=4,2=6]**. The size of an array (i.e. the umber of elements in it) can be computed using the **size** function.

The rules for defining and accessing arrays are:

```
arrterm ::= arrterm '[' tterm ']'
term12 ::= ..
         |  '[' tterm coterm* ']'
```

There is no iteration construct for arrays. The sum of values over an array has to be computed recursively over the size of the array. E.g. using the following definition:

```
fun arrSum(a) = arrIndSum(a,0);
fun arrIndSum(a,i) =
      if (size(a)==i)
      then 0
      else a[i]+arrIndSum(a,i+1)
      fi;
```

4.1.6 User Definitions

The user can define constants and functions in AutoFOCUS. Constants are global (static) definitions that assign values to named constants. For example:

```
const max = 100+50;
```

Functions can be defined using pattern matching. A **pattern** is either a variable, or a constructor (of the appropriate type), applied to the number of patterns the constructor requires. An example for a function with pattern matching is defined over an user defined type:

```
data Store = Empty | Stored(Int,Bool);
fun search(x,Empty) = False
 |  search(x,Stored(y,True)) = (x==y)
```

```
        |  search(x,Stored(y,False)) = (x!=y);
```

The semantics of the functions is that the first "maching" pattern is executed, i.e. the variables in the pattern are bpund to concrete values and the right hand side of the function definition is executed. For example the term **search(1,Stored(2,True))** matches the second definition of **search** with the values **x=1** and **y=2**. The result of evaluating the term is **False** (**1==2**).

The rules for user definitions are:

```
fundecl ::= stereo? funline altline*
         |  stereo? constdef
         |  id ':' type
         |  id ':' type cotype* arrtype+.

constdef ::= 'const' id '=' tterm.
altline ::= '|' line.
line ::= stereo? pat '=' tterm.
tpat ::= pat typean?.
pat ::= id
     |  number
     |  char
     |  string
     |  id '(' tpat copat* ')'.
copat ::= ',' tpat.
```

As can be seen from the grammar also signatures can be declared, e.g. for defining the type of the constant. If signatures are missing, the most general type for the constant/function is inferred. In the above example of max, the inferred type is **NUM**. if this is too general, the type can be restricted to **Int** by

```
max : Int;
const max = 100+50;
```

4.2 Types

Types can be assigned to every term (see Section 4.1). Furthermore model elements (like ports) have types. Similar to terms there can be different types: variables (TVar) constants (TConst) and complex applications (TAppl).

Like terms have types, types have type classes (see Section 1.3). The available type constants are the default types and the user defied types (data definitions, type abbreviations, range and implementation types).

4.2.1 Type Terms

The type terms consist of the special types for sets, arrays and functions. The names of the default and user defined types are the identifiers that can be used in the type terms. An example for a type term is: **`[Bool],Store -> Int`**; If **`Store`** is a user defined type (TConst) this type is used, otherwise a type variable is generated in the type term.

Type terms can be defined using the following rules:

```
type ::= id
       |  id '::' id
       |  '[' type cotype? ']'
       |  '{' type '}'
       |  id '(' type cotype* ')'.
cotype ::= ',' type.
arrtype ::= '->' type.
```

Note that currying is not supported. There are no higher order functions in AutoFOCUS. Therefore the functional type (**`arrtype`**) can only be used once in signature declarations (see Section 4.1.6), but not in arbitrary terms.

4.2.2 Data Definitions

Data definitions are the basis of functional programming languages. They define the identifiers that can be used in type terms and the constructors for the pattern matching in the functions. In addition to the features of many functional languages AutoFOCUS data declarations also define so called selector and discriminator functions for the type.

For example the definition of infinite lists is:

```
data List(a) = Nil | Cons(a,rest:List(a)) deriving ORD;
```

defines not only the constructors **Nil** and **Cons** and the type constructor **List**, but also the following discriminator and selector functions.

```
fun is_Nil(Nil) = True
 |  is_Nil(_) = False;
fun is_Cons(Cons(x1,x2)) = True
 |  is_Cons(_) = False;
fun ConsSel1(Cons(x1,x2)) = x1;
fun rest(Cons(x1,x2)) = x2;
```

The name of the (partial) selector functions can be specified by the user (**rest** in the example). If not specified the default name consists of the Name of the Constructor, "**Sel**" and the number of the position of the selector (e.g. **ConsSel1** for the first selector of **Cons**). Every user defined function that uses pattern matching can be translated into a function that uses discriminators and selectors. This is important for generating code into low level languages like C or Java. The equivalent function to a user defined function can be computed using the API call **Term tFunCase = af.TermUtils.getFunctionTerm(ConstDef);**

The grammar for data declarations is:

```
datadecl ::= stereo? 'data' id datargs? '=' dconstr
       altdconstrs* derivingclass?.
datargs ::= '(' id coid* ')'.
altdconstrs ::= '|' dconstr.
dconstr ::= stereo? id
         |  stereo? id '(' selector? type coseltype* ')'.
coseltype ::= ',' selector? type.
selector ::= id ':'.
derivingclass ::= 'deriving' id coid*.
```

To determine the type class of the type terms like **List(Int)** or **List(Bool)** the **instantiations** of the type constructor into type classes are important. An instantiation consists of a type information and a definition of the characteristic functions of that class. For example, one

could defined the == operation on lists such that all non-empty lists are equal. However in general there exists a "canonical" definition of == that identifies lists, only if they contain the same elements (in the same order). Of course this is only possible if the elements in the list support belong to the class EQ. AutoFOCUS tries to derive such operations (and instantiations) automatically if the declaration has a **deriving** attachment. If subclasses are specified (as ORD in the above example) the super classes are also derived. The default derived order is the order that bases on the order in which the data constructors are specified (e.g. **`Nil < Cons(x,l)`**. If the constructors are identical, the arguments are compared using their order. Arithmetic operations are derived by point-wise extension of the constructor arguments. E.g. the addition on pairs is defined by adding both components in the pair. However this works only for types with exactly one constructor. In the above list example no numeric operations can be derived. In this case the instantiations have to be specified manually.
In the list example this could look as follows:

```
 fun listAdd(Nil,X) = X
 | listAdd(X,Nil) = X
 | listAdd(Cons(x1,r1),Cons(x2,r2)) =
       Cons(x1+x2,listAdd(r1,r2));
instance List(NUM) in NUM where
      fun x + y = listAdd(x,y) and
      fun x - y = listSub(x,y) ann
      fun x * y = listMult(x,y) and
      fun x % y = listMod(x,y) and
      fun x / y = listDiv(x,y);
```

The grammar for type instantiations is:

```
instancedecl ::= 'instance' type 'in' id 'where'
       funinstance andfun*.
andfun ::= 'and' funinstance.
funinstance ::= 'fun' infixline altinfix*.
infixline ::= pat binop pat '=' tterm.
```

```
altinfix ::= '|' infixline.
binop ::= '=' | '==' | '!=' | '<=' | '<' | '>=' | '>' | '+'
        | '-' | '*' | '%' | '/'.
```

The only instances that are allowed are those that instantiate into the known classes (**EQ**, **ORD**, **NUM**). The instances must not be complete, for example the **!=** operation in **EQ** can be derived from **==**, however the arithmetic operations have to be provided completely.

4.2.3 Type Abbreviations

Type abbreviations introduce new names for complex types. For example if the type **[NUM]** is used frequently it can be referred using a symbolic name. This name can be defined by a type abbreviation. For example:

```
type NA = [NUM];
```

The type checker does not differentiate between **NA** and **[NUM]**. However the type of **[1,2]** can only be inferred as **[NUM]** and is not simplified to **NA**. If **[1,2]:NA** is desired it has to be specified.

The grammar for type abbreviations is:

```
typedecl ::= 'type' id datargs? '=' type.
```

4.2.4 Implementation Types

Implementation types are discrete types that are used to implement floating point types on embedded computers that have no floating point units. Implementation types have a fixed size (e.g. 8 bits) and a range (e.g. from -10.0 to 10.0). For fixed size and the range a distance between the values can be computed using the formula *delta = (2^bits -1)/range* Implementation types are integrated into AutoFOCUS to simulate the effect of deploying the software to an embedded target (without floating point unit).

In order to ease the definition and use of implementation types there are two possibilities integrated in AutoFOCUS. It suffices to fix the size or the delta of implementation types. Some examples for the definition of implementation types are:

```
type impl8 = fixed -10.0 to 10.0 size 8;  // inferred:delta
type impl8 = fixed -10.0 to 10.0 delta 0.1;// inferred:size
```

The grammar for the implementation types is:

```
typedecl ::= stereo? 'type' id '=' 'fixed' snumber 'to'
       snumber deltanumber? sizenumber?
deltanumber ::= 'delta' number.
sizenumber ::= 'size' number.
sign ::= '+' | '-'.
snumber ::= sign? number.
```

4.2.5 Range Types

Range types describe subsets of existing types. They consist of upper and lower bounds. For numeric types also a delta can be provided. Some examples for the definition of range types are:

```
type rt = range -1 to 11 delta 2;
type pos = range 1 to 100;
type SomeColors = range Red to Blue;
```

The grammar for the range types is:

```
typedecl ::= stereo? 'type' id '=' 'range' snumber 'to'
       snumber deltanumber?
```

4.2.6 Casts

Casts are ordinary functions in AutoFOCUS that have to be provided, if variables of different types are used. For example (`x:int8+y:int8):int16` is not type correct, but `int8int16(x:int8+y:int8):int16` is correct. For all pairs of numeric types there are casts available (see Section 6.4).

4.3 Modularity (Import)

The grammar for the AutoFOCUS module system (see Section 1.4) is:

```
importdecl ::= 'import' qid
            |  'import' qid '.' '*'.
```

5 Models

The AutoFOCUS parser generates models from the textual syntax. The models are, as all other modeling concepts of AutoFOCUS, specified using a so called meta model. The meta model of AutoFOCUS provides the API to develop new applications. It is described in [AFMM].

This section repeats the meta model of the type system and explains it more detailed.

The overall model is depicted in Figure 9. The most important parts are described in the following subsections.

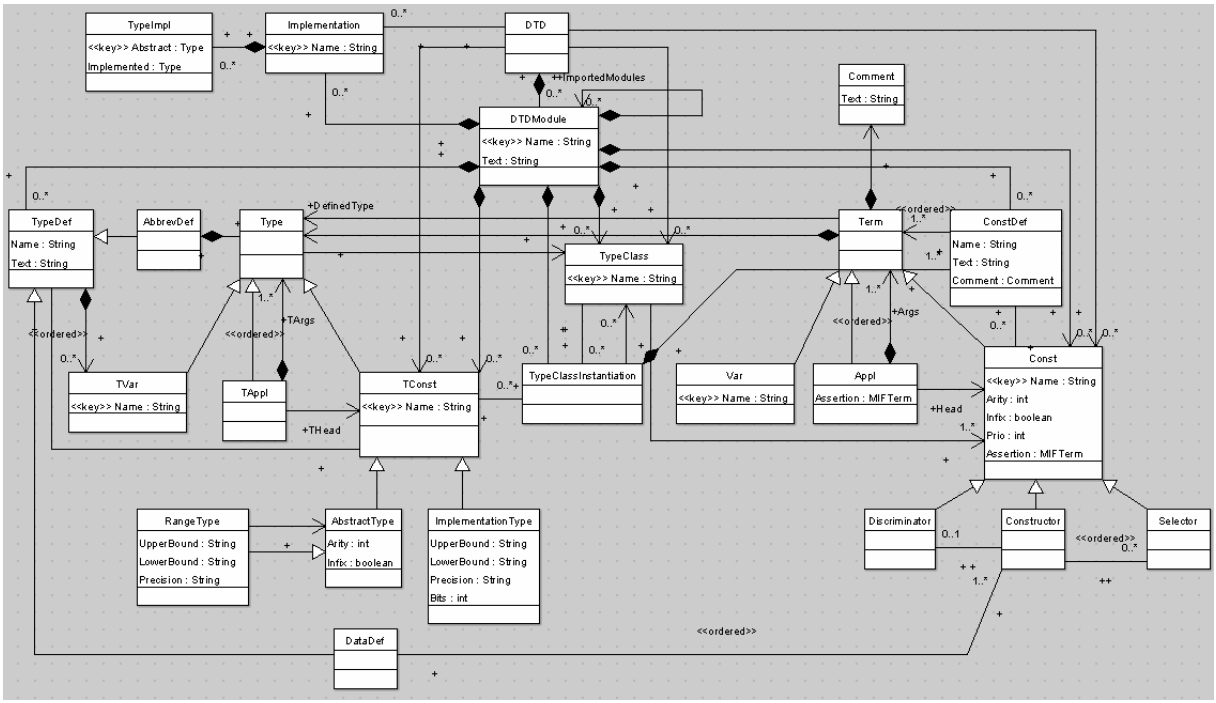

Figure 9: Meta Model of AutoFOCUS Type System

5.1 Terms

The meta model for terms is depicted in Figure 10.

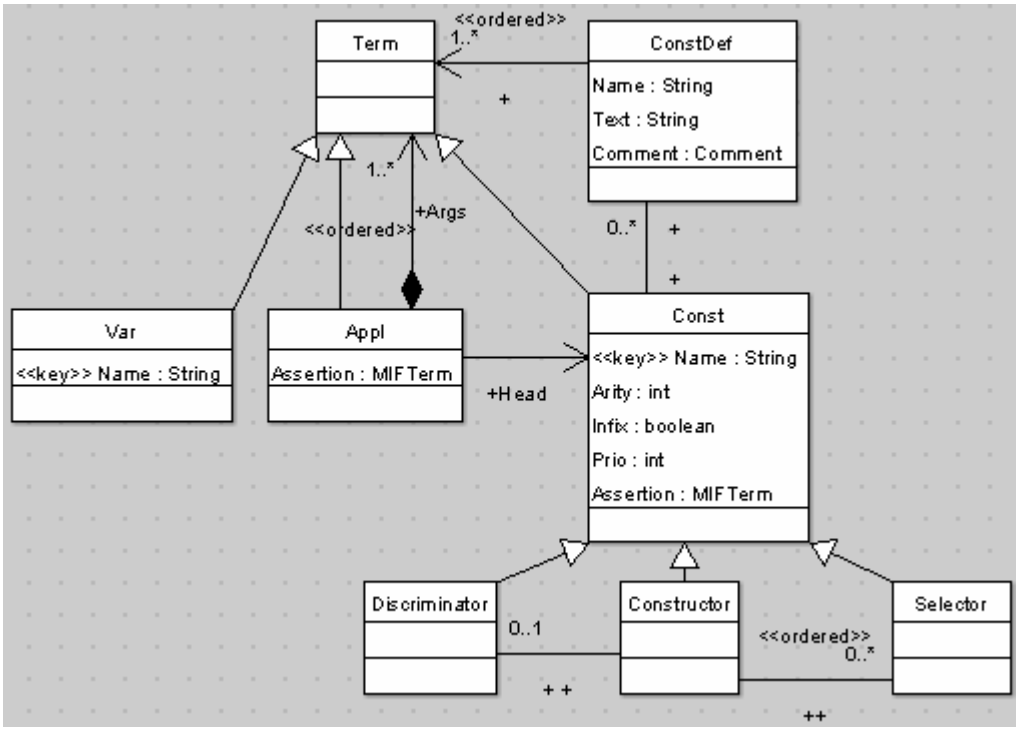

Figure 10: Meta Model of Terms

The elements in the model are:

- **Term** is the abstract class for all terms in AutoFOCUS (see Section 1.1).
- **Var** is the class for variables. Variables are unique in terms, e.g. the **x**'s in **x+2+x** refer to the identical object. Variables have the following attributes:
  - **Name: String** the name of the variable.
- **Appl** is the class for applications. An application has a head-Const and at least one argument. Applications have the following attributes:
  - **Assertion: MIFTerm** the assertion of the application. Note that only applications in the definitions of functions can have assertions that restrict the possible parameter of a function (but do not affect the pattern matching during the evaluation). Assertions of functions can be specified using stereotype. For example by: **fun <<assert: x!=0>> f(x)=1/x**; In this case the term associated with the definition (ConstDef) of f is an Appl and has the assertion associated.

- **Const** describe constants and functions. User defined constants have exactly one definition (ConstDef). For example **`const c = 0.5;`** Examples for constants are "c", "1", "True", and "neg". Using type instantiations (see Section 4.2.2 and 5.3) the predefined constants (like +) can have several ConstDefs (with Type Class instantiations) Constants have the following attributes:
  - **`Name:String`** the name of the constant / functions
  - **`Arity:int`** the number of arguments the constant/function requires
  - **`Infix:boolean`** is true for infix operations like "+"
  - **`Prio:int`** the priority of the operation. Priorities are used for printing the terms without redundant brackets, e.g. **`a+b*c`** instead of **`a+(b*c)`**.
  - **`Assertion: MIFTerm`** is the assertion that is fulfilled of the function or the constant. Assertions to Const can be specified using stereotypes of the form <<assert: term>>. E.g. **`<<assert: f(x)>0>> fun f(x)=x+1;`** If the Const is a Constructor function the assertion restricts the possible arguments. Constructor assertions are specified in the data definition of the constructors, e.g. **`data PosInt = <<assert: num(x)>0>> PN(num:Int);`**
- **ConstDef** represents the definition of user defied constants and functions. ConstDefs have the following attributes:
  - **`Name:String`** the name of the defined constant
  - **`Text:String`** the complete definition of the constants. Comments and formatting information is ignored in this attribute. For an identical text only the Text attribute of DTDModules can be used, since this is set before parsing the module.
  - **`Comment:Comment`** a textual attachment for the definitions. This is used to textually store the stereotypes of the definition.

- **Constructors** are specific constants. Constructors are defined by data definitions. Constructors can be applied in pattern matching definitions. Every constructor with arguments has selector functions (Selector) to select the arguments of the data definition. The number of selector functions is equal to the arity of the constructors.
- **Selector** (see Section 4.2.2). Every selector belongs to it's constructor. One constructor can have several selectors.
- **Discriminator** (see Section 4.2.2) Every discriminator belong to exactly one constructor.

5.2 Types

The meta model of types (and implementation types) is depicted in Figure 11.

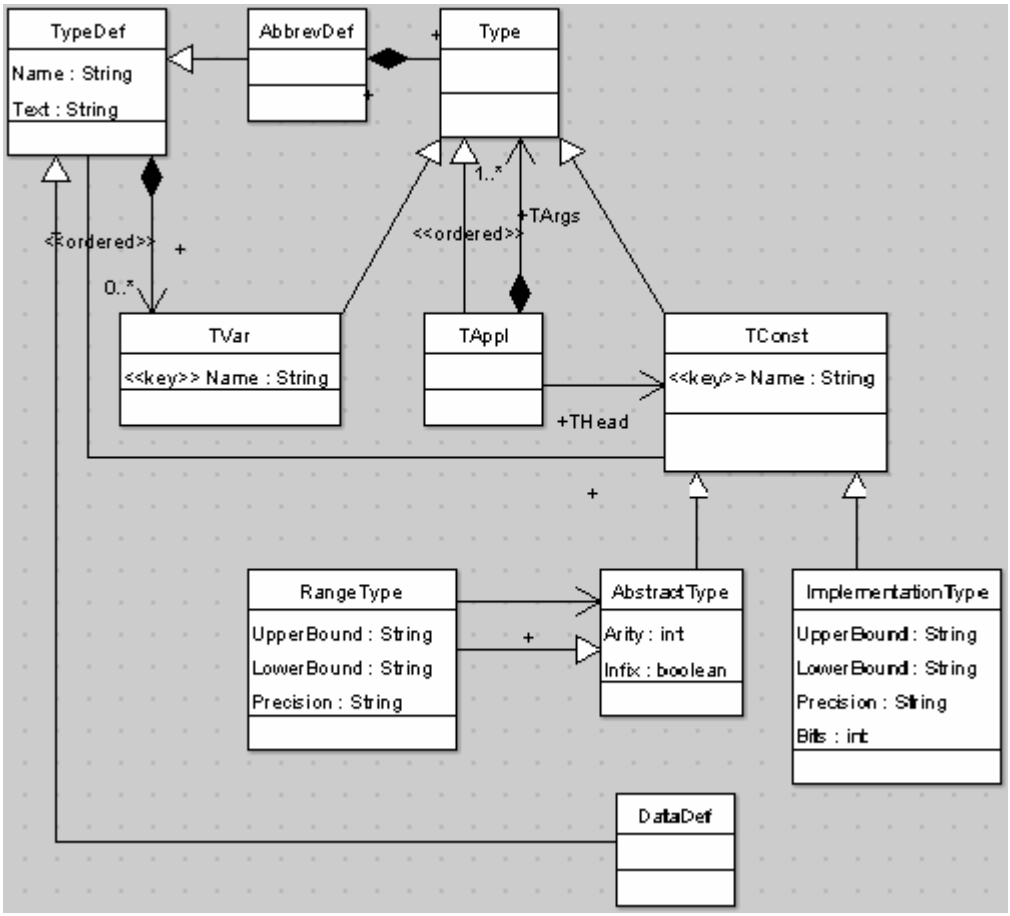

Figure 11: Meta Model of Types

- **Type** is an abstract class for types.
- **TypeDef** is a definition of a user defined type. Type definitions can have type variables if functional types like list(a) are defined. They

have the following attributes

- **`Name:String`** the name of the defined type
- **`Text:String`** the textual representation of the definition without formatting and comments. For an identical text only the Text attribute of DTDModules can be used, since this is set before parsing the module.

- **AbbrevDef** is a special form of type definitions (TypeDef), see Section 4.2.3. The new defined type is a type constant (TConst), possibly with variables. The expanded type is the associated type of the AbbrevDef.
- **DataDef** is a special form of type definitions (TypeDef), see Section 4.2.2. The data definition has associations to the defined constructors.
- **TVar** represents a type variable. By default all types that are not specified will be type variables. TVars are unique in terms, e.g. the **`a`**'s **`x:a+y:a`** refer to the same object. Every type identifier that is not defined (predefined or user defined) will be treated as a type variable. To avoid errors in modeling components due to misspelling of identifiers (e.g. **`imt8`** will be a type variable, since **`int8`** is predefined, but **`imt8`** is not known) type variables in SSDs should start with a quote, e.g. **`'a`**. In DTDs this is not required. TVars have the following attribute:
  - **`Name:String`** the name of the type variable
- **TAppl** (type applications) are type terms, i.e. applications of functional types to type arguments. For example the type **list(a)** is an application of the functional type **list** to the type argument **a**. The relation THead represents the constant head of the term (**list** in the example); the ordered association TArgs represents the arguments of the type application term (**a** in the example).
- **TConst** are the abstract class of atomic parts of types in

AutoFOCUS. Type constants can be abstract types and implementation types. Type constants have the following attributes:

  - **`Name:String`** the name of the type constant

- **AbstractType** represents the unimplemented types of AutoFOCUS. Abstract types can be type constants (or functional types). Type constants can be default type constants like **`Float`**, **`Bool`**, or functional types like **`{.}`**, or **`->`** or user defined types (data and range types). Abstract types have the following attributes:

  - **`Arity:int`** the number of argument the abstract type requires.

  - **`Infix:boolean`** is true for the infix type **`->`**.

- **RangeType** is an abstract type with the description of a range (see Section 4.2.5). A range is defined by a TypeDef. It restricts an abstract type with a lower and an upper bound. RangeTypes have the following attributes:

  - **`UpperBound:Term`** upper bound of the range

  - **`LowerBound:Term`** lower bound of the range

  - **`Precision:Term`** precision (step width) of the range type

- **ImplementationType** is an implemented fixed point term (see Section 4.2.4). Implementation types have the following attributes:

  - **`UpperBound:Term`** upper bound of the range

  - **`LowerBound:Term`** lower bound of the range

  - **`Precision:Term`** precision (step width) of the range type

  - **`Bits:int`** the number of required bits

5.3 Type Classes

The meta model of type classes (and instantiations) is depicted in Figure 12.

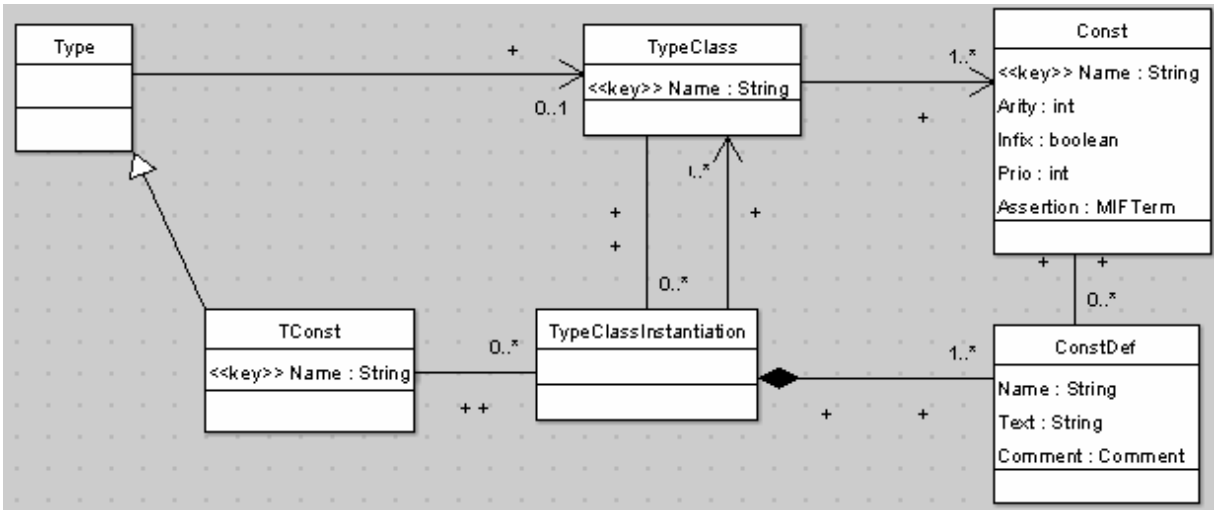

Figure 12: Meta Model of Type Classes

Every type can belong to a type class (EQ, ORD, NUM). The most specific class is noted. If there a no classes available for a type the relation is empty. As described in Section 1.3 and Section 4.2.2 the type class instantiations (TypeClassInstantiation) instantiate a type constructor (TConst) into a type class. Additional requirements on the type classes of the arguments of the type constructor are modeled by the *-association from TypeClassInstantiation to TypeClass. Every TypeClassInstantiation can provide several definitions (ConstDefs) for the overloaded constants (called characteristic constants). Since the predefined constants can have different instantiations every constant can have several ConstDefs. However user defined constants have exactly one ConstDef (without TypeClassInstantiation).

The definition of the meta model of type classes uses the following elements:

- **Type classes** are modeled in TypeClass. Every type class knows it's characteristic operations (*-association to Const). A SuperClass-relation between type classes (as in Haskell) is not needed for AutoFOCUS, since no user defined type classes are supported.
    - **`Name:String`** the name of the type class
- TypeClassInstantiation describes the instantiation of a type into a class.

5.4 Modules

The modules in the type system build the basis of type checking. Every AutoFOCUS component can have it's own DTD with different DTDModules. The DTD is the "symbol table" for parsing terms. All elements in the DTD are unique, such that it suffices to compare the objects instead of the references. The symbol table (DTD) is used for parsing and type checking, for example it contains constants (and types), during parsing these objects are used, it the elements are not found in the DTD variables are generated.

Usually the data types are defined globally, since some mechanisms (like importing) rely on a global unique naming convention of DTDModules.

The DTD contains references to DTDModules, which can contain imported DTDModules. Some Elements (like type definitions) are not used in DTDs, therefore they are only contained in their defining modules.

The meta model for modules is depicted in Figure 13.

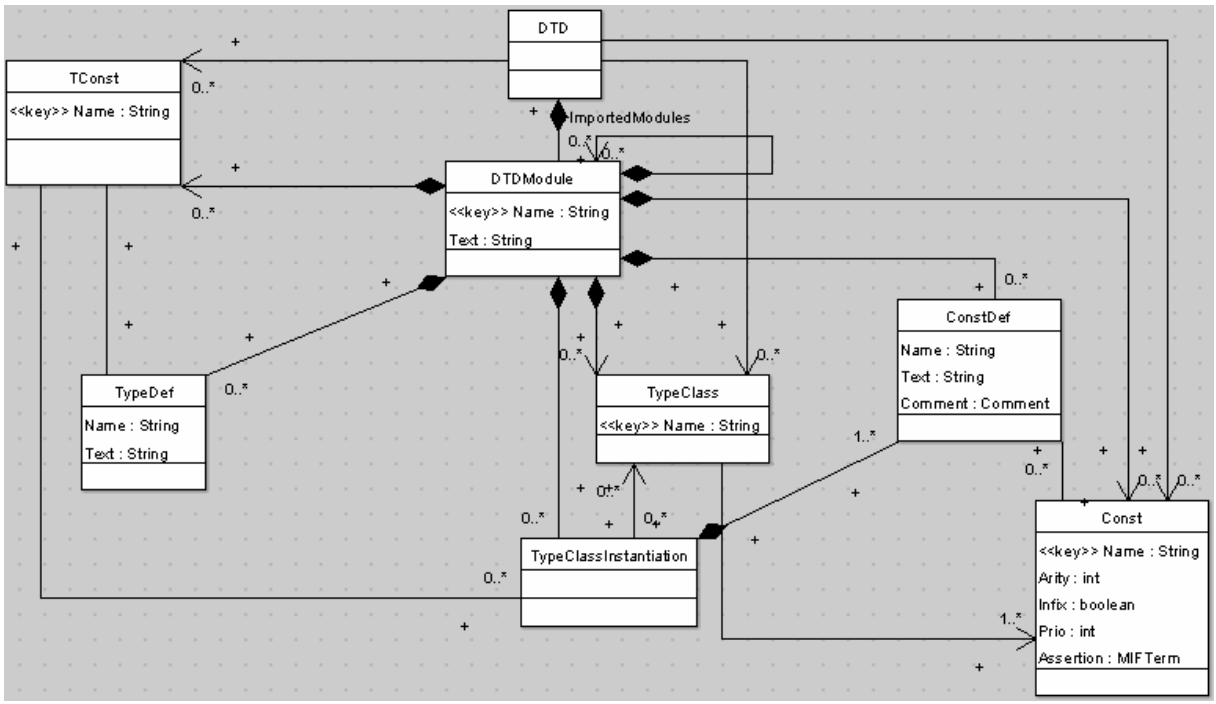

Figure 13: Meta Model of Modules

The used elements of in the meta model of Modules are:

- **DTD** The DTD is the container for all elements. With the indexed associations to type constants, modules type classes and constants (and functions) the DTD is the symbol table of the component and

can be used for determining if the types and terms of the component are correct.

- **DTDModule** is a module of the language. A DTD module corresponds to a file of a programming language, or a DTD element of AutoFOCUS. DTDModules can import other DTDModules. DTDModules contain all constructs of the module. They have the following attributes
    - **`Name:String`** the name of the module
    - **`Text:String`** the textual representation of the module

6 Predefined Elements

The predefined elements of AutoFOCUS can be used in all specification. The realization (definition) is in the class af.SymbolTable which contains the static definitions of the elements.

6.1 Constants

The following constants are predefined in AutoFOCUS:

- Constructor **`True: Bool`**
- Constructor **`False: Bool`**
- Constructor **`NoVal: Channel('a)`**

Note that all constants using the channel type are internal and should not be used in the AutoFOCUS models. For specific applications (like property formulation for model checking) these constants can be helpful.

6.2 Functions

The following functions are predefined in AutoFOCUS:

- Const **`not: Bool->Bool`**
- Const **`size: 'a -> NUM`**
- Const **`let.=.in.`**
- Const **`if.then.else.fi: Bool,'a,'a->'a`**
- Const **`[..]: ['a]`**
- Const **`{.|..}: {'s}`**
- Const **`{}: {'s}`**
- Constructor **`NoVal: 'm -> Channel('m)`**
- Discriminator **`is_Msg: Channel('m)->Bool`**
- Discriminator **`is_NoVal: Channel('m)->Bool`**
- Selector **`Val : Channel('m)->'m`**
- Const **`length : String -> NUM`**
- Const (prio 1) **`=> : Bool, Bool -> Bool`** for implication
- Const (prio 2) **`=> : Bool, Bool -> Bool`** for biimplication

- Const (prio 2) `|| : Bool, Bool -> Bool` for disjunction
- Const (prio 3) `&& : Bool, Bool -> Bool` for conjunction
- Const (prio 4) `bor : NUM,NUM->NUM` for bitwise or
- Const (prio 5) `bxor : NUM,NUM->NUM` for bitwise exclusive or
- Const (prio 6) `band : NUM,NUM->NUM` for bitwise and
- Const (prio 7) `= : 'a,'a->Bool` for equality
- Const (prio 7) `!= : EQ,EQ->Bool` for inequality
- Const (prio 7) `== : EQ,EQ->Bool` for weak equality
- Const (prio 8) `< : NUM,NUM->Bool`
- Const (prio 8) `<= : NUM,NUM->Bool`
- Const (prio 8) `> : NUM,NUM->Bool`
- Const (prio 8) `>= : NUM,NUM->Bool`
- Const (prio 8) `in : 's,{'s}->Bool` for containment in sets
- Const (prio 9) `<< : NUM,NUM->NUM` for left shift
- Const (prio 9) `>> : NUM,NUM->NUM` for right shift
- Const (prio 9) `>>> : NUM,NUM->NUM` for unsigned right shift
- Const (prio 10) `+ : NUM,NUM->NUM`
- Const (prio 10) `- : NUM,NUM->NUM`
- Const `-(.) : NUM->NUM` for unary minus
- Const `+(.) : NUM->NUM` for unary plus
- Const (prio 11) `/ : NUM,NUM->NUM`
- Const (prio 11) `* : NUM,NUM->NUM`
- Const (prio 11) `% : NUM,NUM->NUM`

Some internal constants are available in the symbol table for temporal logic and clock operations.

6.3 Types

The following predefined types exist in AutoFOCUS:

- Abstract Types
  - **Bool**
  - **Int**
  - **Float**

  - **Double**
  - **Char**
  - **String**
  - `{.}` for building sets
  - `[.]` for building arrays
  - `.->.` for building functional types
- Range types
  - **bit** (with values 0 and 1)
  - **int4**
  - **int8**
  - **int16**
  - **int32**
  - **int64**
  - **uint4**
  - **uint8**
  - **uint16**
  - **uint32**
  - **uint64**

Note that since AutoFOCUS uses exact representations of values, there is no difference between **Float** and **Double**, however code generators generate different code (**Float**/**Double**) from them.

6.4 Casts

For all numeric types there exist casting functions in AutoFOCUS (with the expected types).

- Abstract Types
  - **Float2Int**
  - **Int2Float**
  - **Double2Int**
  - **Int2Double**
  - **Double2Float**
  - **Float2Double**

- Range Type: Int4:
  - **int4uint4**
  - **int4uint8**
  - **int4uint16**
  - **int4uint32**
  - **int4uint64**
  - **int4int8**
  - **int4int16**
  - **int4int32**
  - **int4int64**
- the same for int8 to uint64

6.5 Classes

The predefined type classes are mentioned in Section 1.3.

7 Open Issues

No import for components (hence all elements are imported, and overloading is resolved arbitrarily if no qualification is used). Sets (and manual type class instantiations) are currently not completely implemented as described. Type abbreviations are currently not checked correctly.

8 References

[AF] The tool AutoFOCUS, available at http://autofocus.in.tum.de

[AF2] The tool AutoFOCUS2, available at http://www4.in.tum.de/~af2/

[Haskell] The programming language Haskell, available at http://www.haskell.org

[AFMM] Description of the AutoFOCUS meta model and it's API. Included in the distribution of AutoFOCUS in <AFHOME>/usermanuals/ValidasAPI.pdf

A. AutoFOCUS Grammar

The complete Grammar (see Help of DTD-Editor) is:

```
main ::= maindeclsemi*.
maindeclsemi ::= maindecl ';'.
maindecl ::= datadecl
           |  typedecl
           |  impldecl
           |  importdecl
           |  instancedecl
           |  fundecl
           |  arrterm.
qid ::= id
     |  qualid.
coid ::= ',' id.
number ::= decimal
         |  hexnumber.
sign ::= '+'
       |  '-'.
snumber ::= sign? number.
temporal ::= '<>'
           |  '()'
           |  '[]'.
tempneg ::= temporal
          |  neg.
sampleinit ::= 'init' term1.
importdecl ::= 'import' qid
             |  'import' qid '.' '*'.
instancedecl ::= 'instance' type 'in' id 'where'
        funinstance andfun*.
andfun ::= 'and' funinstance.
funinstance ::= 'fun' infixline altinfix*.
infixline ::= pat binop pat '=' tterm.
altinfix ::= '|' infixline.
binop ::= '='
```

```
        |  '=='
        |  '!='
        |  '<='
        |  '<'
        |  '>='
        |  '>'
        |  '+'
        |  '-'
        |  '*'
        |  '%'
        |  '/'.
datadecl ::= stereo? 'data' id datargs? '=' dconstr
       altdconstrs* derivingclass?.
datargs ::= '(' id coid* ')'.
altdconstrs ::= '|' dconstr.
dconstr ::= stereo? id
         |  stereo? id '(' selector? type coseltype* ')'.
coseltype ::= ',' selector? type.
selector ::= id ':'.
derivingclass ::= 'deriving' id coid*.
typedecl ::= stereo? 'type' id '=' 'range' snumber 'to'
       snumber deltanumber?
          |  stereo? 'type' id '=' 'fixed' snumber 'to'
       snumber deltanumber? sizenumber?
          |  stereo? 'type' id datargs? '=' type.
deltanumber ::= 'delta' number.
sizenumber ::= 'size' number.
impldecl ::= stereo? 'implementation' id '=' implterm
       addimplterm*.
implterm ::= '[' ']'
          |  '[]'
          |  id
          |  '[' idecl coidecl* ']'.
coidecl ::= ',' idecl.
idecl ::= type '->' type.
addimplterm ::= '+' implterm.
fundecl ::= stereo? funline altline*
```

```
          |  stereo? constdef
          |  id ':' type
          |  id ':' type cotype* arrtype+.
funline ::= 'fun' line.
constdef ::= 'const' id '=' tterm.
altline ::= '|' line.
line ::= stereo? pat '=' tterm.
tpat ::= pat typean?.
pat ::= id
     |  number
     |  char
     |  string
     |  id '(' tpat copat* ')'.
copat ::= ',' tpat.
tterm ::= interm typean?.
interm ::= arrterm 'in' arrterm
        |  arrterm.
arrterm ::= arrterm '[' tterm ']'
         |  term0.
term0 ::= term1 '|_|' term0
       |  term1 '$AU' term0
       |  term1 '$EU' term0
       |  term1.
term1 ::= term2 '=>' term1
       |  term2 '<=>' term1
       |  term2 'when' term1
       |  term2 'sample' term2 sampleinit?
       |  term2 'default' term1
       |  term2.
term2 ::= term3 '||' term2
       |  term3 'fby' term2
       |  term3.
term3 ::= term4 '&&' term3
       |  term4.
term4 ::= term5 'bor' term4
       |  term5.
```

```
term5 ::= term6 'bxor' term5
        |  term6.
term6 ::= term7 'band' term6
        |  term7.
term7 ::= term8 '=' term8
        |  term8 '!=' term8
        |  term8 '==' term8
        |  term8.
term8 ::= term9 '<' term9
        |  term9 '<=' term9
        |  term9 '>' term9
        |  term9 '>=' term9
        |  term9.
term9 ::= term10 '<<' term9
        |  term10 '>>' term9
        |  term10 '>>>' term9
        |  term10.
term10 ::= term11 '+' term10
         |  term11 '-' term10
         |  term11.
term11 ::= term12 '*' term11
         |  term12 '/' term11
         |  term12 '%' term11
         |  term12.
term12 ::= sign? tempneg* idorsize '(' tterm coterm* ')'
         |  'if' tterm 'then' tterm 'else' tterm 'fi'
         |  'let' id '=' arrterm 'in' term12
         |  sign? tempneg* '(' tterm ')'
         |  '[' tterm coterm* ']'
         |  '{' '}'
         |  '{' tterm coterm* '}'
         |  '{' tterm '|' tterm coterm* '}'
         |  sign? tempneg* qid '@'?
         |  '@'
         |  char
         |  string
```

```
        |  sign? neg? number
        |  'case' tterm 'then' tterm 'else' tterm 'esac'
        |  'merge' tterm 'then' tterm 'else' tterm 'egrem'.
coterm ::= ',' tterm.
idorsize ::= id
          |  'size'.
typean ::= ':' type.
type ::= id
      |  id '::' id
      |  '[' type cotype? ']'
      |  '{' type '}'
      |  id '(' type cotype* ')'.
cotype ::= ',' type.
arrtype ::= '->' type.
// important lexical helpers:
////////////////////////////
decimal ::= digit* '.'? digit+.
stereo ::= '<<' ( not_gt* ( not_gt '>' not_gt )* )* not_gt*
       '>>'.
string ::= '"' not_quote* '"'.
char ::= ''' ( not_char | '\n' | '\0' | '\t' | '\\' ) '''.
hexnumber ::= ( '0x' | '0X' ) ( digit | [ 'A' .. 'F' ] | [
       'a' .. 'f' ] )+.
neg ::= '!' | 'not'.
qualid ::= letter ( letter | alpha | '.' )*.
id ::= dollar* letter alpha*.
```